

# Resolução do Labyrinth Robots utilizando Métodos de Pesquisa em Linguagem C++ (Tema 7/Grupo 39)

João Álvaro Ferreira (up201605592)  
Mestrado Integrado em Engenharia  
Informática e de Computação  
Faculdade de Engenharia e da  
Universidade do Porto  
Porto, Portugal  
up201605592@fe.up.pt

João Augusto Lima (up201605314)  
Mestrado Integrado em Engenharia  
Informática e de Computação  
Faculdade de Engenharia e da  
Universidade do Porto  
Porto, Portugal  
up201605314@fe.up.pt

João Carlos Maduro (up201605219)  
Mestrado Integrado em Engenharia  
Informática e de Computação  
Faculdade de Engenharia e da  
Universidade do Porto  
Porto, Portugal  
up201605219@fe.up.pt

**Resumo**—O artigo em questão contém uma pequena descrição do jogo Labyrinth Robots, de como o jogar e uma formulação da resolução deste puzzle com métodos de inteligência artificial. Para além disto, apresentaremos também representações do puzzle (tanto nossas como as originais), trabalho já desenvolvido, conclusões que tirámos e algumas referências e trabalhos relacionados que nos foram úteis.

**Keywords**—Inteligência Artificial, Labyrinth Robots, Algoritmo A\*, Pesquisa em Profundidade, Pesquisa em Largura, Pesquisa Gananciosa.

## I. INTRODUÇÃO

No primeiro projeto da unidade curricular de Inteligência Artificial, do MIEIC, foi-nos proposto abordar um de vários puzzles pré-definidos e obter uma resolução para este. O jogo cuja solução vamos explorar é denominado de Labyrinth Robots. O jogo consiste na navegação de um labirinto numa grelha ortogonal, por múltiplos robôs, sendo que o objetivo do jogo é que cada robô chegue a uma célula de destino específica..

## II. DESCRIÇÃO DO PROBLEMA

O jogo Labyrinth Robots, por Emil Fridell, é um jogo de puzzle inspirado no jogo de tabuleiro Ricochet Robots. Este género de jogos prende-se na navegação de um labirinto com vários atores, sendo que estes têm de ser combinados de formas inteligentes para chegar ao objetivo.



O problema prende-se na descoberta do caminho que leve à solução com menos movimentos. Os robôs terão de interagir uns com os outros com movimentos em ordens específicas, dependendo do nível, de modo a chegar ao estado final com o menor número de movimentos possível. Para realizar um movimento, o jogador começa por seleccionar o robô que pretende mover e, após isso, selecciona uma direção ortogonal - esquerda, direita, cima ou baixo. Tendo escolhido tanto o robô como a direção, o robô movimenta-se até chegar a um obstáculo - seja esse obstáculo uma parede ou outro robô.



De importante mencionar também que cada robô tem a sua célula final específica, e que muitas vezes os robôs poderão ser apenas mais uma obstrução ou a única forma de chegar ao resultado final.

## III. FORMULAÇÃO DO PROBLEMA

Representação de estados:

Os estados são representados numa matriz de caracteres de tamanho variável. Os caracteres em questão são "X" para as paredes, " " para espaço vazio, [a-z]\{x} para representar os robôs e [a-z]\{x} para a posição objetivo de cada robô. Os robôs estão posicionados consoante as suas coordenadas

```

C X X X X X X X X X X X X X X X X
X X X X X X X      X X X      X X X X
X X X              X          X X X
X X               X            X X X
X X       X X           X        X
X X     X X X             X      X
X         X                X
X                    X X          X
X                   X X X         X
X X X              X X X         X
X X               A              X
X X   B    X          X X        X
X         X          X           X X
X X       X                  X X X
X X X              X           X X X
X X X              X           X X X
X X X X X X X X X X X X a C X X X X X
X X X X X X X X X X X X X X X X X X

```

Cada letra maiúscula (exceptuando o X, que representa paredes) é um robô, sendo as minúsculas as respectivas posições objetivo.

Os operadores são verificados nas funções MoveTop, MoveBottom, MoveLeft e MoveRight que simulam um movimento, e nas funções PlayTop, PlayRight, PlayLeft e PlayBottom, que os executam, sendo todas estas de player. Estas funções são chamadas pelo jogador ou pelos algoritmos de pesquisa, sendo que em ambos os casos o custo é tomado em conta pela função do algoritmo (no caso de jogar o PC) ou pelo loop de jogo (no caso do player), chamando a função makeMove. A verificação do fim do jogo é feita na função checkEndGame. Todas estas funções são parte da classe Player.

O construtor da classe Maps é responsável pela leitura dos tabuleiros dos puzzles a partir de ficheiros txt com formatos específicos, sendo que está pré-determinado para ler de um ficheiro chamado "maps.txt", tendo esse um formato específico. Estes mapas, e os estados de jogo, são imprimidos no ecrã (como demonstrado anteriormente) pela função printMap da classe mapa.

Para validar jogadas/operadores temos o loop de jogo, na função de mesmo nome da classe Game, que utiliza a função makeMove para verificar a validade de cada jogada a nível de sintaxe, servindo-se das funções de Move e Play anteriormente mencionadas para verificar a validade segundo as regras do jogo.

Servimo-nos também da computeSolution para chamar as funções dos algoritmos e, tendo encontrado uma solução, fazer o display dos movimentos feitos.

Para a heurística, servimo-nos das funções computerHeuristic, optimistic e realistic, e uma variável HEURISTIC para determinar o tipo de heurística escolhido pelo utilizador.

## VI. ALGORITMOS DE PESQUISA

Os algoritmos de pesquisa utilizados neste projeto foram o Depth-First Search, o Breadth-First Search, A-Star, Iterative Deepening Depth-First Search e Greedy. Implementaríamos separadamente também o algoritmo de Custo Uniforme, mas devido ao problema em questão ser representado com um custo de 1 para cada jogada, este é idêntico ao Breadth-First Search.

Os resultados de todas as pesquisas são guardados num vetor de pares best\_move, que contem todos os movimentos da resposta obtida.

O algoritmo de Depth-First Search foi implementado recursivamente. Uma função auxiliar é utilizada para o utilizador escolher o nível limite da profundidade de pesquisa (o limite para o algoritmo fazer backtracking portanto) e uma função principal percorre os movimentos em profundidade, utilizando recursão. Para guardar as posições visitadas é utilizada uma estrutura map, devido à rapidez de acesso, mas esta funcionalidade é opcional devido ao quanto esta pode atrasar a pesquisa.

O algoritmo de Breadth-First Search foi implementado com o auxílio de uma estrutura Node, sendo que esta será utilizada também em todos os seguintes algoritmos. Esta estrutura contém informações sobre o movimento/jogada

que esta representa, contendo o movimento em si, a posição dos robôs, o mapa do jogo e um apontador para o nó pai. No algoritmo BFS, são gerados os nós da posição inicial (os movimentos possíveis a partir desta) e colocados numa queue, sendo que sempre que estes são avaliados e saem da queue, os seus respectivos filhos são adicionados ao fim. Desta forma, existe uma pesquisa em largura das jogadas possíveis.

O algoritmo A-Star adiciona os filhos do Node inicial a um set de Nodes, escolhendo o próximo Node a partir desse set. Este processo repete-se, tomando em conta o G (custo total) e H (peso com base na heurística) de cada node, até encontrar a solução ótima.

O algoritmo Greedy funciona de forma idêntica ao A-Star, mas o valor de G é sempre 0.

O Iterative Deepening Depth-First Search é uma versão iterativa do algoritmo Depth-First Search em que o limite de nível da pesquisa é aumentado progressivamente (após todos os nós terem sido visitados) até chegar a uma solução ou ao limite final estipulado pelo utilizador.

## VII. EXPERIÊNCIAS E RESULTADOS

De modo a testar a eficiência dos algoritmos utilizados, traduzimos alguns dos níveis do jogo original para a nossa versão. Para compararmos as diferenças entre os algoritmos dependendo da situação, os níveis tentam explorar uma grande diversidade de dificuldade de resolução, sendo que os níveis analisados são o nível 1, 5, 10, 15, 20 e 25 (o último do jogo). Não só a complexidade dos mapas aumenta, como também o número de robôs e objetivos, frequentemente implicando um maior número de jogadas. É importante mencionar também que, no puzzle em questão, os movimentos são iguais ao custo da solução.

Obtivemos também o número de movimentos da solução mínima do jogo, que oferece este valor como pista. Para o Algoritmo A-Star, usamos os valores da heurística mais eficiente para o nível em questão.

Nível 1 - 1 robot, 1 goal (Solução Ótima: 9 moves)

Algoritmo	Nódulos Percorridos	Movimentos	Tempo(s)
Depth-First Search	24	12	0.0016648
Breadth-First Search	26	9	0.0024328
A-Star	16	9	0.0009308
Iterative Deepening Depth-First Search	242	9	0.0174531
Greedy	16	9	0.0010453

Nível 5 - 2 robots, 2 goals (Solução Ótima: 6 moves)

Algoritmo	Nódulos Percorridos	Movimentos	Tempo(s)
Depth-First Search	333	19	0.0292015
Breadth-First Search	34	0 (Failed)	0.0058958
A-Star	24	6	0.004263
Iterative Deepening Depth-First Search	1759	6	0.225353
Greedy	10	6	0.0012584

Nível 10 - 2 robots, 2 goals (Solução Ótima:12 moves)

Algoritmo	Nódulos Percorridos	Movimentos	Tempo(s)
Depth-First Search	18504	19	2.13125
Breadth-First Search	68	0 (Failed)	0.0150353
A-Star	353	9	0.0690994
Iterative Deepening Depth-First Search	?	? (Failed)	? (Failed)
Greedy	257	135	0.0710082

Nível 15 - 3 robots, 3 goals (Solução Ótima:13 moves)

Algoritmo	Nódulos Percorridos	Movimentos	Tempo(s)
Depth-First Search	?	? (Failed)	?
Breadth-First Search	113	0 (Failed)	0.0262483
A-Star	20026	13	526.903
Iterative Deepening Depth-First Search	?	? (Failed)	? (Failed)
Greedy	244	116	0.125232

Nível 20 - 3 robots, 3 goals (Solução Ótima:12 moves)

Algoritmo	Nódulos Percorridos	Movimentos	Tempo(s)
Depth-First Search	?	? (Failed)	?
Breadth-First Search	58	0 (Failed)	0.0112261
A-Star	694	12	0.360836
Iterative Deepening Depth-First Search	?	? (Failed)	? (Failed)
Greedy	152	53	0.0406479

Nível 25 - 4 robots, 3 goals(Solução Ótima:24 moves)

Algoritmo	Nódulos Percorridos	Movimentos	Tempo(s)
Depth-First Search	?	? (Failed)	?
Breadth-First Search	207	0 (Failed)	0.0539749
A-Star	?	? (Failed)	? (Failed)
Iterative Deepening Depth-First Search	?	? (Failed)	? (Failed)
Greedy	6946	3810	328.429

Nos resultados observados, verificamos que depth-first search é útil para encontrar uma solução rápida, embora não a mais otimizada em níveis com poucos robôs e movimentos, mas que é, até na velocidade, inferior a outros algoritmos. O algoritmo de breadth-first search leva-nos à solução mais otimizada para problemas cuja melhor solução tem poucos movimentos e apenas um robô, mas que mal um destes valores aumente o branching factor torna-se demasiado e o programa é levado ao falhanço.

O A-Star é, ao todo, o algoritmo mais consistente a encontrar a melhor solução no menor tempo possível, encontrando frequentemente soluções com menos movimentos do que os da melhor solução sugerida pelo jogo original. No entanto, o último puzzle torna-se demasiado complexo e o A-Star é incapaz de o resolver num tempo razoável, sendo o único em que não é melhor opção (comparando os tempos) para obter a melhor solução.

O algoritmo Iterative Deepening Depth-First Search obtém a melhor opção de forma iterativa, percorrendo uma

quantidade de nódulos muito superior aos outros algoritmos, mas isto leva a um tempo de cálculo também muito superior. Apesar de obter as melhores soluções nos dois primeiros problemas, é inferior em tempo ao A-Star e ao Greedy, e falha nos restantes.

O algoritmo Greedy é o único que foi capaz de obter uma solução em todos os níveis, apesar de esta solução ser frequentemente muito superior à ótima. No último puzzle, em que é o único a obter uma solução, a solução em questão tem um custo de 3538, estando longe da solução ideal com um custo de 24. Para os restantes níveis, é inferior a nível de custo às restantes opções bem sucedidas, sendo no entanto melhor a nível de tempo.

## VIII. CONCLUSÕES E PERSPETIVAS DE DESENVOLVIMENTO

O projeto em questão é uma modulação bem sucedida do jogo Labyrinth Robots com a implementação de soluções tanto pelo utilizador como por algoritmos de pesquisa. Desenvolver este projeto permitiu-nos expandir o nosso conhecimento sobre formulação de problemas, sobre os algoritmos utilizados e as várias possíveis implementações destes, já que bastante esforço e experimentação foi feito de modo a otimizar as respostas obtidas o máximo possível.

Os resultados obtidos foram satisfatórios, apesar de não terem sido completamente bem sucedidos. Alguns dos algoritmos, nomeadamente o Breadth-First-Search e o Iterative Deepening Depth-First Search, foram incapazes de obter a solução ótima em níveis em que consideramos que seriam capazes de o fazer. Esperávamos também que o algoritmo A-Star fosse superior aos restantes em todos os resultados, sendo que essa expectativa se concretizou em

todos os níveis menos no último, que o A-Star não conseguiu resolver. Em alguns casos, as nossas expectativas foram ultrapassadas, quando a solução sugerida pelo jogo original tinha um número de jogadas superior à obtida pelos algoritmos.

Futuramente, gostaríamos de poder adicionar uma interface gráfica ao trabalho, para o tornar mais fácil de utilizar, e tanto otimizar mais os algoritmos já implementados como possivelmente adicionar outros.

## REFERÊNCIAS BIBLIOGRÁFICAS

Michael Fogleman, “Ricochet Robots: Solver Algorithms”, 2012, [online], Disponível em:

<https://speakerdeck.com/fogleman/ricochet-robots-solver-algorithms>, consultado em Março 2019.

Amit , “Pathfinding”, 2011, [online], Disponível em: <http://theory.stanford.edu/~amitp/GameProgramming/>, consultado em Março 2019.

Antti Laaksonen, Competitive Programmer’s Handbook, Edição 3 de Julho, 2018.

Damian Barczyński, A-Star Implementation, , 11 de Outubro de 2017 , Disponível em: [https://github.com/daancode/a-star/blob/master/source/AStar.cpp?fbclid=IwAR0xWPwJ1\\_KvejikLaiwvfE57arCwXLdmQyb9uEF\\_LZs7sgB3Q19xLpD8iU](https://github.com/daancode/a-star/blob/master/source/AStar.cpp?fbclid=IwAR0xWPwJ1_KvejikLaiwvfE57arCwXLdmQyb9uEF_LZs7sgB3Q19xLpD8iU)

Luis Paulo Reis, Apontamentos de Inteligencia Artificial 2018/2019